Microcontrollers

AppNote AP3232

☐ or additional file
uCOS2_TC10GP_REL1P0.exe available

MicroC/OS-II

The Real-Time Kernel

Ported to the Infineon TriCore TC10GP (TriBoard Platform)

MicroC/OS-II which stands for *Micro-Controller Operating System* (µC/OS-II) is a real-time kernel with performance comparable to many commercially available kernels. The internals of µC/OS-II are described in the book by Jean J. Labrosse entitled *MicroC/OS-II, The Real-Time Kernel*, ISBN 0-87930-543-6. This book contains ALL the source code for µC/OS-II. A port for the Infineon TriCore processor is available both from Infineon and the official µC/OS-II web site: www.uCOS-II.com.

Thousands of people around the world are using µC/OS in all kinds of applications such as cameras, medical instruments, musical instruments, engine controls, network adapters, highway telephone call boxes, ATM machines, industrial robots, and many more. Numerous colleges and Universities have also used µC/OS and now µC/OS-II to teach students about real-time systems.

Andre Gompel / Microcontroller & DSP Applications, Infineon US, San Jose

Table of Contents

1.0 INTRODUCTION 3
1.1 PREREQUISITE 4
1.2 µC/OS-II FEATURES 4
1.3 µC/OS-II LICENSING POLICY 5
1.4 µC/OS-II WEB SITE 6
2.0 TriCore PORT ASSUMPTIONS 7
3.0 DEVELOPMENT TOOLS 8
4.0 BUILDING AND RUNNING THE CODE 9
4.1 STEP 0: Setup and Documentation 9
4.2 STEP 1: Extract the source 9
4.3 STEP 2: Copy debugger script 9
4.4 STEP 3: Load and Run (Using the provided executable) 9
4.4 STEP 4: Building the code 10
4.5 STEP 5: Description of uCOS2test.c 11
5.0 CODE AND IMPLEMENTATION DESCRIPTION 13
5.1 Initialization 13
5.2 Traps 13
5.3 Interrupts. 13
5.4 Fast Context switching 13
5.5 Task switching 14
5.6 Critical sections access and lock 14
5.7 Creating a Task 14
5.8 Non Standard Processor Independent Code (PIC) 14
5.9 Details of programming 15
5.9.1 TriCore specific macros (tc10gp.h) 15
5.9.2 Startup file crt0.tri 15
5.9.3 OSInitTriCore() 15
5.9.4 Modification to µC/OS-II's OSInit() 15
5.9.5 Task stack pointer (A10) 16
5.9.6 Task context registers 16
5.9.7 Interrupt stack pointer 16
5.9.8 Memory mapping and EBU registers settings 16
5.9.9 Building ROMable code (for Flash or EPROM) 16
5.9.10 Osprintf() 17
5.9.11 µC/OS-II tick 17
5.9.12 Code size 17
5.9.13 Makefile and build process 18
6.0 ADDENDUM: makefile, and uCOS2.C listings 19
7.0 BIBLIOGRAPHY 26
8.0 Web Sites 27
9.0 ACKNOWLEDGEMENTS 27

1.0 INTRODUCTION

MicroC/OS-II (also known as µC/OS-II) is a Real-Time Operating System (RTOS) written by Jean J. Labrosse. Labrosse wrote an early version of this RTOS for *Embedded Systems Programming* (ESP) magazine back in 1992. The article described this RTOS (called µCOS) and was published in two consecutive months (May and June 1992). The article was one of the most popular articles published by ESP. Later that year, Labrosse expanded upon the article and introduced his first book called *µC/OS, The Real-Time Kernel* published by R&D Books. This book sold well over 15,000 copies all around the world. The second edition of µC/OS was introduced in 1999 and consisted of an almost totally rewritten book. This book is called *MicroC/OS-II, The Real-Time Kernel* and is now considered a classic for anyone who wants to learn about Real-Time Operating Systems.

Sudhir Verman, manager of the RTOS activities for Infineon asked me to do the "**MicroC/OS-II**" port to the **Infineon TriCore** "**TC10GP**". Initially, this was a background activity (i.e. part time), and the first version was done for an early version silicon of TriCore (called Rider A). Because the port was a 'spare time' activity, it took nearly six weeks to get a functional version.

Then the "real" silicon came out and contained some changes, although minor from the programmer's standpoint. The RTOS port was then finalized for use with a board called the **TriBoard** developed by Infineon in Munich, Germany.

Doing the TriCore port of MicroC/OS-II has been an enjoyable experience. I hope you will enjoy using it. If you need an RTOS, you may want to consider µC/OS-II because it's robust, has a small footprint, you get the source code and the book explains its architecture and use.

Since all the source code is provided with the µC/OS-II book, you can modify it to suit your own needs. The book and the source code are good complements to the TriCore documentation.

This is the first release of the TriCore port. One of the goals was to make the port simple. The main objective was to get good performance (especially on task switching by the scheduler), make the port straightforward and easy to understand, and therefore allow you to modify it to your needs.

Every attempt has been made to make the TriCore port consistent with Jean Labrosse's implementation: no function has been renamed and pretty much everything works as documented in Jean Labrosse's book (with the exception that TriCore is a context rather than a stack oriented processor).

This release of the TriCore port does not take full advantage of some of the advanced features of the TriCore architecture. To fully take advantage of some of the TriCore features would require changes to µC/OS-II's processor independent code. So far, attempts have been made to make minimal changes to this part of the code.

1.1 PREREQUISITE

This application note assumes a basic understanding of real-time kernels and that you are familiar with C. I also assume that you have access to Jean J. Labrosse's book. Without the book, you may not understand some of µC/OS-II's subtleties and this would make your understanding of the TriCore port more difficult. If this is your first contact with an RTOS, you should read the first few chapters of this book.

1.2 µC/OS-II FEATURES

Source Code:
The µC/OS-II book contains ALL the source code for this RTOS. Many commercial real-time kernels are provided in source form. You will have a hard time finding source code that is as neat, consistent, well commented and organized as µC/OS-II's. Giving you the source code is not enough. You need to know how the code works and how the different pieces fit together. You will find this type of information in the book. The organization of a real-time kernel is not always apparent by staring at many source files and thousands of lines of code.

Portable:
Most of µC/OS-II is written in highly portable ANSI C, with target microprocessor-specific code written in assembly language. Assembly language is kept to a minimum to make µC/OS-II easy to port to other processors.

ROMable:
µC/OS-II was designed for embedded applications. This means that if you have the proper tool chain (i.e. C compiler, assembler and linker/locator), you can embed µC/OS-II as part of a product.

Scalable:
µC/OS-II was designed so that you can use only the services that you need in your application. This means that a product can have just a few of µC/OS-II's services while another product can have the full set of features. This allows you to reduce the amount of memory (both RAM and ROM) needed by µC/OS-II on a product per product basis. Scalability is accomplished with the use of conditional compilation. You simply specify (through **#define** constants) which features you need for your application/product. µC/OS-II was designed to reduce both the code and data space.

Preemptive:
µC/OS-II is a fully preemptive real-time kernel. This means that µC/OS-II always runs the highest priority task that is ready. Most commercial kernels are preemptive and µC/OS-II is comparable in performance with many of them.

Multi-tasking:
µC/OS-II can manage up to 64 tasks, however, the current version of the software reserves eight (8) of these tasks for system use. This leaves your application with up to 56 tasks. Each task has a unique priority assigned to it. This means that µC/OS-II cannot do round robin scheduling. There are thus 64 priority levels.

Deterministic:
Execution time of all µC/OS-II functions and services are deterministic. This means that you can always know how much time µC/OS-II will take to execute a function or a service. Furthermore, except for one service, execution time of all µC/OS-II services does not depend on the number of tasks running in your application.

Services:
µC/OS-II provides a number of system services such as mailboxes, queues, semaphores, fixed-sized memory partitions, time related functions, etc.

Interrupt Management:
Interrupts can suspend the execution of a task and, if a higher priority task is awakened as a result of the interrupt, the highest priority task will run as soon as all nested interrupts complete. Interrupts can be nested up to 255 levels deep.

Robust and reliable:
µC/OS-II is based on µC/OS which has been used in hundreds of commercial applications since 1992. µC/OS-II uses the same core and most of the same functions as µC/OS yet offers many more features. µC/OS-II has attained FAA DO-178B level B certification and will soon attain level A certification.

1.3 µC/OS-II LICENSING POLICY

µC/OS-II is a great real-time kernel and has proven itself in countless applications all around the world.

µC/OS-II's source and object code can be freely distributed (to students) by accredited Colleges and Universities without requiring a license, as long as there is no commercial application involved. In other words, no licensing is required if µC/OS-II is used for educational use.

You MUST obtain an **'Object Code Distribution License'** to embed µC/OS-II in a product. In other words, you must obtain a license to put µC/OS-II in a product that is sold with the intent to make a profit. There will be a license fee for such situations and you will need to contact Micrium, Inc. (see below) for pricing.

You MUST obtain an **'Source Code Distribution License'** to distribute µC/OS-II's source code. Again, there will be a fee for such a license and you will need to contact Micrium, Inc. (see below) for pricing.

For licensing information send an e-mail to:

Licensing@Micrium.com

Or write to:

Micrium, Inc.
949 Crestview Circle
Weston, FL 33327
U.S.A.

+1 954 217 2036
+1 954 217 2037 (FAX)

1.4 µC/OS-II WEB SITE

The official µC/OS-II WEB site (www.uCOS-II.com) contains the following information:

- News on µC/OS-II,
- Availability of ports to other processor architectures,
- Answers to frequently asked questions (FAQs),
- Application notes,
- List of recommended books,
- List of classes (courses) being offered,
- Links to other WEB sites, and
- More.

2.0 TriCore PORT ASSUMPTIONS

- The software has been tested on the TriBoard with the TC10GP processor which contains 4 Mbytes of SDRAM.

- The TriCore integrated GPTU is used to generate the RTOS interrupt ticks. This time source allows kernel services to implement time delay functions and timeouts on certain calls.

- I/O is accomplished using a function called `OSprintf()`, which is reentrant. Mutual exclusion to this function is accomplished by using a semaphore.

  `OSprintf()` uses an RS232 port which is configured to 9600 bps (Bits-Per-Second), 8 bits, no parity, 1 stop bit and no hardware handshake. Transmit and Receive is interrupt-driven but can easily be modified to accommodate your own requirements.

  The RS232 port uses one of the two TriCore integrated UARTs, called ASC0 (see the Infineon TC10GP User's Manual).

3.0 DEVELOPMENT TOOLS

The TriCore port along with µC/OS-II was compiled using the Green Hills Software tools for TriCore (TC10GP) Assembler/Compiler/Linker ver 1.8.9 with January 2000 patches. The code size is approximately 15 Kbytes.

In order to modify and rebuild the code you need:

1) The Green Hills Software toolchain for TriCore.
2) A 'make' utility.
3) The Signum Chameleon version 2.50 (or later) debugger which is used to download the code to the TriBoard.
4) The following utilities which are also needed and invoked from the makefile. These are available from the MKS Toolkit, which is available for Windows NT.

```
grep
sed
rm
```

You can use non-EABI compliant C compilers with the TriCore port but this may require some rewriting of the low-level code. This is due to the fact that the parameter passing (both ways) between the C code and the assembly code assume the content and availability of certain registers.

4.0 BUILDING AND RUNNING THE CODE

For this port, the Windows NT environment is assumed. Although the `C:` drive is used, the build is not drive-specific. Also note that you can run the code without re-building it, as the executable compiled with debug options is provided in the file `uCOS2test.tce`.

4.1 STEP 0: Setup and Documentation

a) Verify that the TriBoard TC10GP's on-board oscillator (on 14-pin DIP socket) is marked 16.0000 MHz.

b) You have the Green Hills Software toolchain installed (version 1.8.9 with patch, or newer).

c) The `make` and `rm` utilities should be available in your path. These are only required to re-build the code, not to run it.

d) The Signum Chameleon debugger version 2.50 or later should be installed.

e) You have a copy of Jean J. Labrosse's book, *MicroC/OS-II, The Real-Time Kernel* (if you want reference to the source code).

4.2 STEP 1: Extract the source

Execute the provided self-extracting zip file, uCOS2_TC10GP_REL1P0.exe. Assuming your installation drive is C:, this archive will create the following directory structure:
`C:\software\uCOS-II\TC10GP\Ghs\Source\` for the source code and
`C:\software\uCOS-II\TC10GP\Ghs\Doc\` for the documentation files.
Although not a requirement, this structure is recommended by Jean J. Labrosse. When you are done, you will have all the required files, including one or more files with a `.tce` extension (which stands for "TriCore Executable"), in the folder containing the source code.

4.3 STEP 2: Copy debugger script

Copy the startup script `TB_TC10GP_4M_SDRAM.mac` from the 'Doc' folder into, the Signum executable directory, which is usually `c:\Signum\Chameleon` to the file `tricore.mac`.

This script may be useful to initialize the TriCore `ADDRESS` and `BUSCON` registers.

4.4 STEP 3: Load and Run (Using the provided executable)

a) Make sure that the Signum debugger works properly with the board. If necessary use Signum technical support. You should be able to see the CPU registers, etc. Click on the top left icon and load the file `uCOS2test.tce`.

b) Connect the TriBoard DB9 connector (9 pins, next to the printer connector) to a "dumb terminal". I have been using the Windows NT hyper-terminal, configured as:

- COM1:
- 9600 Baud
- 8 bits
- No parity
- No hardware handshake
- VT100 emulation.

c) Check that the TriBoard DIP switches are set as follows:

- SW1 OFF
- SW2 ON
- SW3 ON
- SW4 ON
- SW5 OFF
- SW6 OFF
- SW7 ON
- SW8 ON

d) Check that the TriBoard 'Jumpers' are set as follows:

- JP401 ON (towards the board's edge)
- JP402 ON (towards the board's edge)
- JP501 ON (towards the board's edge)

e) Run the code by clicking on the RUN icon. The dumb terminal should display a message and task 1 should be running continuously. Once in a while, TASK #2 (which is waiting for a semaphore) will display a message due to the semaphore time-out. If you press a key on the 'dumb' terminal, the ASCO receive interrupt will signal the semaphore by calling the µC/OS-II function `OSSemPost()`. This action will wake up the task.

4.4 STEP 4: Building the code

The makefile provided with the TriCore port will work without modifications if the Green Hills toolchain is properly installed (assuming you installed the compiler on the `C:` drive). Also, the file `Regdefb.h` (in the 'Doc' folder) has been updated and copied to `c:\green\tria\include`. The next release of the Green Hills tools should integrate these changes.

Important: The toolchain needs a patch that was released in early January 2000.

Type: `make`

The build process should be clean, i.e., no errors, no warnings. The file created is called `uCOS2test.tce` (TriCore Executable). The build creates other files: `.o` (object files), `.map` (linker map file) and `.lst` (listing files)

Now load the file `uCOS2test.tce` as explained in STEP 6 above.

4.5 STEP 5: Description of uCOS2test.c

As with most C programs, the code starts in `main()`. `main()` starts by calling `OSInitTriCore()` to initialize processor specific registers, interrupt vectors, the serial I/O port and more (see `OS_CPU_C.C`). As with all µC/OS-II applications, `main()` needs to call `OSInit()` to initialize µC/OS-II's data structures. `main()` then creates a semaphore which will be used to wake up a task upon receiving a character from the dumb terminal. `Main()` then starts `AppStartTask()` and the calls `OSStart()`. µC/OS-II will start `AppStartTask()` because that's the only task created in `main()`. `AppStartTask()` starts off by printing:

```
=== Starting MicroC/OS-II, The Real-Time Kernel, Infineon Tricore Version ===
    RTOS by Jean Labrosse, TriCore port by Andre Gompel, (Size 15 Kbytes)
 In this application, a main task creates two tasks, then starts them.
Task 1 executes once, then yields time, then Task 2 executes and waits for its
semaphore (initialized to 0) to be released by a "Post" or "V" primitive when a
character is received by the on-chip UART (ASC0).

 A task counter is incremented each time the task "task1" is awakened.
 ________ Now Running ________
```

`AppStartTask()` then calls `OSStatInit()` to initialize µC/OS-II's statistics task which is used to compute available CPU time. `AppStartTask()` then creates two application tasks: `AppTask1()` and `AppTask2()`. When the two tasks are created, `AppStartTask()` prints out:

```
 Starting AppStartTask infinite loop
```

`AppStartTask()` then enters an infinite loop which basically does nothing except run every clock tick and gives up the CPU. Alternatively, this task could have deleted itself (i.e. terminated itself).

`AppTask1()` starts off by printing:

```
 Task 1, TaskCtr = 0
```

`AppTask1()` then enters an infinite loop, increments a task counter, displays its value as well as one of four character: `-`, `\`, `|` or `/` which is basically showing a 'rotating wheel':

```
 Task 1 pass 1    -
```

`AppTask1()` then calls `OSTimeDlyHMSM()` to delay the task for 2 seconds. At this point, µC/OS-II suspends execution of `AppTask1()` until the desired delay expires. After 2 seconds, µC/OS-II returns to `AppTask1()` and toggles the on-board LED.

When µC/OS-II first executes `AppTask2()`, this task displays:

```
Task 2 now waiting for an ASC0 interrupt to release the semaphore.  Pass # 0
```

`AppTask2()` then increments a counter and waits for the semaphore to be signaled or a timeout to expire after 10,000 ticks. If you type a character at the dumb terminal or nothing during the time it takes to count 10,000 ticks then, `AppTask2()` will display:

```
Task 2 waken (pass # ??) up by a receive interrupt or semaphore timed out
```

The "`??`" above shows where the counter value will actually appear. `AppTask2()` then calls `OSTimeDly(1)` to delay execution for one tick. This function is not actually necessary but shows that you could add additional code and other service calls to µC/OS-II if needed.

Most of the time (over 99%), µC/OS-II is running its idle task. In other words, the above application requires so little CPU time that the CPU has nothing to do and waits for something to happen - time tick or a character to be received on the RS-232C port.

We suggest that you try modifying `uCOS2test.c` to add your own code. You should, however, gain a good understanding of µC/OS-II before you do so.

5.0 CODE AND IMPLEMENTATION DESCRIPTION

5.1 Initialization

MicroC/OS-II was written for a stack-oriented machine. TriCore is different (see TriCore architecture manual). TriCore uses a linked list in memory called context save areas. This is a very elegant and powerful feature of the TriCore architecture. However, it diverges from a stack-oriented machine to the point that some minor changes to µC/OS-II's processor independent code (PIC) have to be made. For this reason, the file `os_core.c` required changes.

5.2 Traps

The `BTV` (Base Trap Vector register) is initialized in the startup module `crt0.tri`. It probably does not need to be changed. However, if you do modify the `BTV` initialization, make sure that it points to valid memory.

At this time the trap handler is very crude. For a real application, some code needs to be added for trap recovery. This may come in a subsequent version of the TriCore port.

5.3 Interrupts.

Interrupt vectors are statically set. The code is found in the file `os_cpu_a.tri`. I did not see the need to increase code size, but if you need it, writing a function to dynamically hook an interrupt node to an interrupt is a simple exercise. However, the exception processing must be handled carefully.

Currently the interrupt priorities are:

2 for the Timer (`GPTU`) tick.
3 for the Transmit interrupt
4 for the Receive interrupt.

At task creation, the `CCPN` (CPU Current Priority Number) is set to `1`, which will allow any of the above hardware interrupts to win arbitration. If you need to change them, edit the code in `os_cpu_a.tri`.

5.4 Fast Context switching

TriCore was designed for very fast context switches. A context switch takes two clock cycles to save or restore all the TriCore context registers (one per context). This includes all 16 data registers and most (but not all) of the 16 address registers.

Read the TriCore Architecture Manual Section 4, "Managing Tasks and Functions" to learn more about this feature.

5.5 Task switching

This code is very efficient since it takes advantage of the TriCore architecture. A task switch is normally triggered by means of a *system call*. The related code is in `os_cpu_a.tri`, and also `os_cpu_c.c`.

Some further optimizations of the RTOS would be possible, but have not been attempted so far. This is possible because TriCore has some built-in features which have not been used by this port. This may allow to further reduce context switching, reduce the code size, and decrease interrupt latency. The optimizations have not been performed to avoid changing µC/OS-II's processor independent code.

5.6 Critical sections access and lock

The simplest way to enter a critical section is described by Jean Labrosse (`OS_CRITICAL_METHOD` number 1). This simply consists of disabling interrupts before entering a critical section and re-enabling upon exiting the critical section.

It has some annoying side effects which are described in Labrosse's book (see Section 9.03.02).

`OS_CRITICAL_METHOD` number 2 (also see section 9.03.02) which consist of saving and then restoring the `ICR` register. This was the method chosen because it prevents running into the problems described in the book.

5.7 Creating a Task

Only the function `OSTaskCreateExt()` is supported. In other words, you MUST NOT use `OSTaskCreate()`. `OSTaskCreateExt()` does everything `OSTaskCreate()` does and more. `OSTaskCreate()` is a legacy function from the previous version of µC/OS (i.e. the predecessor of µC/OS-II).

Furthermore, porting code from µC/OS will not be a problem, just replace the call to `OSTaskCreate()` by a call to `OSTaskCreateExt()` and supply the additional arguments. Note that in the code I have used, the task ID is the same as the task priority number. Also take note that priority number `0` is the highest priority number while priority `63` is the lowest.

5.8 Non Standard Processor Independent Code (PIC)

Jean Labrosse intended to have most of the code in µC/OS-II processor independent. For the TriCore port, I had to slightly modify `OS_CORE.C`. The other PIC files are untouched. The TriCore specific files for the port are `os_cpu_c.c`, `os_cpu_a.tri`, `tc10gp.h` and `rename_f.h`.

The following functions have been changed in `OS_CORE.C`:

`OSIntEnter()`
`OSIntExit()`
`OSTCBInit()`
`OSInit()` (added a call to `OSInitHook()`)

The changes are minor and documented in the code. The changes are mostly due to the context-oriented architecture of TriCore. Also the `GW` bit has to be set. In ISRs, it's being cleared by default.

5.9 Details of programming

In order to view the files properly, set the TAB character to 4 spaces in your favorite editor. Also, the files are DOS format, meaning that each line ends with a `CR` (Carriage Return, `0x0D`) and `LF` (Line Feed, `0x0A`) instead of only a `CR` as found in UNIX-based editors.

5.9.1 TriCore specific macros (tc10gp.h)

The macro `OS_ENABLE_GW()` enables the access to global registers. It is used because after entering an ISR, the `PSW GW` bit is disabled. And we need access to global registers. Note that this macro should be used whenever entering an ISR, which will later (within the ISR) make use of the access to critical sections, macros `OS_ENTER_CRITICAL` and `OS_EXIT_CRITICAL`.

5.9.2 Startup file crt0.tri

The startup file `crt0.tri` has the same name and functionality as the library provided file. However it is very different, so no attempt should be made to use the GHS library file with same name. It would not work, unless modified. It has to be noted that GNU uses this name for startup modules.

5.9.3 OSInitTriCore()

`OSInitTriCore()` is provided to initialize the TriCore (TC10GP) peripherals. This function is called by `main()` and MUST precede the call to `OSInit()`.

5.9.4 Modification to µC/OS-II's OSInit()

`OSInit()` has been changed to add a call to a new function called `OSInitHook()` (see `OS_CPU_C.C`). `OSInitHook()` simply creates a semaphore needed by `OSprintf()` to ensure mutual exclusion to the serial output port. You should note that the creation of the semaphore could also have been done in `main()` after the call to `OSInit()`. This would avoid having to change µC/OS-II's processor independent code.

5.9.5 Task stack pointer (A10)

Each task has its private stack that is statically initialized at task creation. Note that the stack pointer in this implementation complies with the TriCore EABI requirement: for a "push", the stack pointer is decremented, for a "pop", the stack pointer is incremented. This implies that the stack pointer for each task is initialized to point to the last word (or higher word address) of the task stack.

5.9.6 Task context registers

Each task has its private context registers: `FCX`, `PCXI` and `LCX`.

5.9.7 Interrupt stack pointer

ISRs use the interrupt stack pointer initialized in the startup module `crt0.tri`.

5.9.8 Memory mapping and EBU registers settings

The current build assumes that the code runs in segment 11 (`0xBXXXXXXX`). The debugger script does the following initialization:

```
EBUCON  = 0x0000FF68
ADDSEL0 = 0xA0000061
ADDSEL1 = 0xB0000051
ADDSEL2 = 0x00000001
ADDSEL3 = 0x00000000
DRMCON1 = 0x00000023
BCUCON  = 0x4001FFFF
BUSCON0 = 0x00420A7C
BUSCON1 = 0x01023000
BUSCON2 = 0x80415FFF
BUSCON3 = 0x80415FFF
DRMCON0 = 0x1F000030
```

5.9.9 Building ROMable code (for Flash or EPROM)

If you want to "burn" the code in flash, you need to set `ADDSEL1` to `0xA0000051`. This will run the code from the flash, starting at address `0xA0000000`, which is the TC10GP reset address. You may also need to disable any external memory segment which are not used. Set the corresponding address register LSB to `0`.

For the code to run, you need the ENDINIT bit to be set (TRUE). Refer to a recent version of the documentation (1999 and later) - there have been some changes made. You also need to disable the watchdog timer by setting WDTCON bit 3 to TRUE. Then you need to modify the linker/locator file (TriBd_RB.lnk) and map the code in segment 10 (0xA0000000). Refer to the documentation for GHS or whatever linker you are using. "Flashable" code can be created after a successful "make" by typing "make flash". The resulting file is named uCOS2test_flash.tce. At the time of this writing, this has not be fully tested (but the generated code mapped in segment 10 will run from RAM, the EBU initialization code need to be added and tested.)

5.9.10 Osprintf()

The function OSprintf() is reentrant, and is added to the RTOS for Triboard. It is fully compatible with the ANSI C standard library.

5.9.11 µC/OS-II tick

The tick source for µC/OS-II uses the TriCore on chip timer GPT0 (general-purpose timer 0). See the user's manual for the TC10GP. Only one 16-bit timer is used and is connected to SysClk. All other timers are available to the user. Note that the timer's tick function may result in a task preemption, so it is recommended to avoid modifying timer 0 code.

5.9.12 Code size

The code size for the sample code which includes µC/OS-II has been optimized for size and does not contain any debug information. The code size (the .text segment) is about 15 Kbytes. Smaller code size could be expected with newer versions of the compiler or a different compiler. Below is the output from the linker.

```
.spad              4  0x00000004
.contexts         12  0x0000000C
.intstack        384  0x00000180
.traps           256  0x00000100
.interrupts      256  0x00000100
.text          14464  0x00003880
.syscall           8  0x00000008
.data             12  0x0000000C
.bss           24524  0x00005FCC
.heap          65536  0x00010000
.stack          5120  0x00001400
.rodata         1164  0x0000048C
.romdata          12  0x0000000C

Total:        111752  0x0001B488
```

The code quality of the compiler is quite good. However, if the size of the code is an issue, you can remove some of the features in µC/OS-II (see `OS_CFG.H`) which may save you a few kilobytes. In this case, I would suggest a look at Jean Labrosse's book.

Programming with µC/OS-II is very straightforward. At the time of this writing, there is no known deviation from Jean Labrosse's book.

The example code `uCOS2test.c` is simplistic. If you wish to rebuild it under a different name, just change it in the makefile (see `PROJ`, line 16). It is recommended to redo the build in a dedicated directory where you can expand the `.tgz` file prior to modifying the makefile.

5.9.13 Makefile and build process

You need to read this section only to modify the code. Familiarity with the structure of a makefile is assumed.

Note that lines 30 and 33 have the `G` and `g` flags for building with debug information. In order to do a build without debug information, just comment out the `-WA,-` and `-G` by placing a `'#'` to their left.

Also, if your GHS toolchain is not under the default configuration, you may have to modify line 12.

To build the code, just type `make` (this assumes that the make utility is in the 'path'). It is also assumed that `rm`, `tar` and `chmod` are available.

Type: `make clean` to delete all the object files.

Type: `make flash` to build the code for flash. This will create the file `uCOS2test_flash.tce` mapped in segment 10 (`0xA0000000`). Note that, at the time of this writing, the generated code has only been tested in RAM and lacks some register initialization (EBU registers mostly).

Type: `make` to rebuild the code (after a `make clean` or modification of one of the source files).

Type: `make archive` to recreate an archive with all the files encapsulated in a file with the name of the project and a `.tgz` extension. Note that the output file will be write protected, therefore, for success you may have to delete, rename or unprotect the previously made archive file.

Note that the released `makefile` may be slightly different from the file listed below.

6.0 ADDENDUM: makefile, and uCOS2.C listings

```
#
# Created : mid 99 Andre Gompel, initially for TriCore Rider A
# Updated for TC10GP / Rider B on TriBoard with TC10GP chip.
#
# Question(s): email me at agompel@bigfoot.com and/or Andre.Gompel@Infineon.com
# Several updates: used Green Hill toolchain 1.8.9 with 1/2000 patch.
#
# For more details, read attached documentation.
#

UNIX_TOOLS    =  /opt/xvw_1.0r0     # Rider_B   # For Unix build GHS path
NT_GHS_TOOLS =  c:/green           # Default installed path for GHS on Windows NT
UX_GHS_TOOLS =  /opt/ghstrl_1.8.9
#GPATH        =  c:/cygnus/cygwin-b20/H-i586-cygwin32/bin/

PROJ     =  uCOS2test         # modify your project name: example test for test.c
SIGDIR   = c:/Signum/Chameleon

TOOLS    = $(NT_GHS_TOOLS)
BIN      = $(TOOLS)
CC       = $(TOOLS)/cctri     # C compiler
INCDIR   = $(TOOLS)/libsrc    # Mostly header files
LIBDIR   = $(TOOLS)/tri       # Compiler Liraries

LNK_FILE = TriBd_RB.lnk
TRICORE_DEFS = tc10gp.h

COMOPTS = -list -dual_debug -dwarf -nostrip -cpu=riderb :startfile_dir=.

ASFLAGS = $(COMOPTS) -I$(INCDIR) -c -DEMBEDDED -DACODE  -Wa,-g

CFLAGS  = $(COMOPTS) -slashcomment -c -list -noasmwarn -pragma_asm_inline \
          -passsource -D__LANGUAGE_C   -OS   -G  # -nosym

LFLAGS  = $(COMOPTS) $(LNK_FILE) -lnk=-map

LFLAGS_F= $(COMOPTS) TriBdFlash.lnk -lnk=-map


ALLOBJS = $(PROJ).tce

INCLUDES = includes.h os_cfg.h os_cpu.h uCOS_II.h Tc10GP.h

OBJS     =  $(PROJ).o  os_core.o os_cpu_a.o os_cpu_c.o os_mbox.o
            os_mem.o os_q.o os_sem.o os_task.o os_time.o

FILES    =  $(PROJ).c makefile $(LNK_FILE)    ..\Doc\
       crt0.tri  os_cpu_a.tri os_core.c  os_cpu_c.c \
       os_mbox.c os_mem.c os_q.c os_sem.c os_task.c os_time.c \
       includes.h os_cpu.h uCOS2test.h  \
       os_cfg.h $(TRICORE_DEFS) ucos_ii.h lcd.h rename_f.h
       $(PROJ).tce $(PROJ)_flash.tce

.SUFFIXES : .tri .c .o .tce

.c.o:
    $(CC) $< $(CFLAGS)

.tri.o:
    $(CC) $< -o $*.o $(ASFLAGS)

all         : $(PROJ).tce

crt0.o      : $(TRICORE_DEFS) crt0.tri
```

```
65  os_cpu_a.o  : $(TRICORE_DEFS) os_cpu_a.tri
66
67  $(PROJ).o   : $(INCLUDES) $(PROJ).c  rename_f.h # example uCOS2test.c
68  os_core.o   : $(INCLUDES) os_core.c  rename_f.h
69  os_cpu_c.o  : $(INCLUDES) os_cpu_c.c
70  os_mbox.o   : $(INCLUDES) os_mbox.c
71  os_mem.o    : $(INCLUDES) os_mem.c
72  os_q.o      : $(INCLUDES) os_q.c
73  os_sem.o    : $(INCLUDES) os_sem.c
74  os_task.o   : $(INCLUDES) os_task.c
75  os_time.o   : $(INCLUDES) os_time.c
76
77  $(PROJ).tce : $(OBJS) crt0.o $(LNK_FILE)
78              $(CC) -o $*.tce $(LFLAGS) $(OBJS)
79
80  flash:
81      $(CC) -o $(PROJ)_flash.tce $(LFLAGS_F) $(OBJS)
82
83
84  clean:              # type "make clean" to clean all output files.
85      rm -f $(ALLOBJS) $(OBJS) *.lst *.dbg *.idb *.sym *.stf
86
87
88  archive:            # type make archive to create an archive.
89      tar cvfz $(PROJ).tgz $(FILES)
90      chmod -w $(PROJ).tgz
91
92  #
93  # To restore the archive file.tgz  use "tar xvfz filename where filename will be
94  # uCOS2test.tgz if you do not modify this file.
95  # To Build the code, extract the tgz file, the type make
96  # This assumes that the GHS toolchain has been installed on drive C:
97  # This makefile is simple and starightforward, but if you need to learn more
98  # read the O'Reilly book "make": a good reference.
```

Jan 24 20:27 2000 [1]uCOS2test.c Page 1

```
 1  /*********************************************************************************************
 2  *                                                uC/OS-II
 3  *                                          The Real-Time Kernel
 4  *                           (c) Copyright 1998, Jean J. Labrosse, Plantation, FL
 5  *                                           All Rights Reserved
 6  *
 7  * File          :  TEST.C
 8  * By            :  Jean J. Labrosse
 9  * Modified by   : Andre Gompel      for Infineon TriCore implementation of UCOS-II
10  *                 Andre.Gompel@Infineon.com agompel@bigfoot.com
11  *
12  *                 Connect TriBord DB9 connector to a "dumb" terminal set at
13  *                 9600 bds, no parity, 8bits, no H/W flow control.
14  *
15  *
16  *********************************************************************************************/
17
18  #include "INCLUDES.H"
19  #include "uCOS2test.h"
20  #include "rename_f.h"
21
22  OS_STK  AppStartTaskStk[256];   // Word aligned, implicit
23  OS_STK  AppTask1Stk[256];
24  OS_STK  AppTask2Stk[256];
25  OS_EVENT *RxSem;                        // Receive Semaphore
26  extern OS_EVENT *pSem;
27
28
29  static void  AppStartTask(void *pdata);
30  static void  AppTask1(void *pdata);
31  static void  AppTask2(void *pdata);
32  static void  AppTickInit(void);
33
34
35  INT8U MyDbg;
36
37  os_tsk_ext_t        tsk_ext;
38  p_os_tsk_ext_t      p_tsk_ext = &tsk_ext;
39
40  void main (void)
41  {
42      char message[] = " Test of the transmit routine";
43      char *p_msg = (char*) &message;
44
45                                                      // Initialize Context globals,
46      OSInitTriCore();                                // Initialize Tricore registers,
47                                                      // Initialize Tricore timers.
48
49
50     OSInit();    /* Init defaults, create "OSTaskIdle", may create"OSSTASKStatStk" */
```

[1] Note that for space reason, only the main program is listed, also look at file "**os_cpu_c.c**" for the ASC0 ISR, the complete code listing would be quite bulky, and also may not be up to date.

```
       Jan 24 20:27 2000  uCOS2test.c  Page 2

51
52       RxSem = OSSemCreate(0); // Create receive Semaphore, itialized to 0.
53                               // The first "P" primitive or OSSemPend()
54                               //  will put task in the task waiting list.
55                               // REF: J.L's book page 306.
56
57
58    /*
59     We may want to create the task, only if the context and stack
60     requirements have been fullfilled.
61     Need more work for dynamic task creation.
62
63    In fact OsTakCreateExt is a macro which first initialize the CSA list for the task,
64    then call "normal" OsTaskCreateExt.
65    It was implemented that way to keep the processor independent code
66    untouched.
67    */
68
69        OSTaskCreateExt(AppStartTask,                           // p_task code
70                                (void *)0,                      // pdata not used here
71                                &TaskStartStk[TASK_STK_SIZE-1],// p_tos     (top of stk)
72                                TASK_1_PRIO,                // Task priority Zero=Highest
73                                 TASK_1_PRIO,                  // Task ID
74                                &TaskStartStk[0],             // p_bos
75                                TASK_STK_SIZE,                // n of stack words
76                                (void *)p_tsk_ext,            // p_Context info
77                         OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR); // Task options
78
79        /*---- Create any other task you want before we start multitasking -----------*/
80
81        OSStart();              /* Start multitasking (i.e. give control to uC/OS-II) */
82    }
83
84   /*********************************************************************************
85    *                                          STARTUP TASK
86    *
87    * Description : This is an example of a startup task.  As mentioned in the book's text, you MUST
88    *               initialize the ticker only once multitasking has started.
89    * Arguments   : pdata   is the argument passed to 'AppStartTask()' by 'OSTaskCreate()'.
90    * Notes: 1) The first line of code is used to prevent a compiler warning because 'pdata' is not
91    *                   used.  The compiler should not generate any code for this statement.
92    *     2) Interrupts are enabled once the task start because the I-bit of the CCR register was
93    *                   set to 0 by 'OSTaskCreate()'.
94
*************************************************************************************************/9
95
96
97    /*
98    **
99    **  Andre's note: the tasks priorities are higher when their number is lower,
100   ** also some priorities are reserved by the RTOS: see JL book, page 77, 3.01 Tasks.
```

```
Jan 24 20:27 2000  uCOS2test.c  Page 3


101   **
102   ** In order to create a task, the call to os_ctx_alloc must return a non null
103   ** value thru the global variable g_tsk_ctx0.
104   ** OSTaskCreate first will call for context allocation.
105   **
106   **
107   */
108
109   static void  AppStartTask (void *pdata)
110   {
111        INT8U first_time=TRUE;
112
113        OSprintf("\x1b[2J \n\r=== Starting MicroC/OS-II, The Real-Time Kernel, Infineon TriCore
      version ===");
114        OSprintf("\n\r RTOS by Jean Labrosse, TriCore port by Andre Gompel,   (Size 15
      Kbytes)\n\r");
115
116   OSprintf
116        ("\n\r In this application, a main task creates two tasks, then start them");
117
118
119   OSprintf
120       ("\n\r Task 1 execute once, then yield time, the Task 2 execute and wait its semaphore \
121   (initialized to 0) to be released by a \"Post\" or \"V\" primitive when\
 122    a character is received by the on chip UART (ASC0)\
 123    \n\n\r A task couter is incremented each time the task \"task1\" is waken up\
124       \n\r __________   Now Running  __________");
125
126
127        OSStatInit();
128
129       OSTaskCreateExt(AppTask1,                                  // p_task code
130                               (void *)0,                         // pdata not used here
131                               &TaskStartStk[TASK_STK_SIZE-1],// p_tos     (top of stk)
132                               TASK_2_PRIO,                       // Task priority
133                              TASK_2_PRIO,                       // Task ID
134                               &TaskStartStk[0],                  // p_bos
135                               sizeof(AppTask1Stk)/sizeof(OS_STK),          // n of stack words
136                               (void *)p_tsk_ext,                              // p_Context info
137                         0);                                                   // Task options
138
139
140       OSTaskCreateExt(AppTask2,                                          // p_task code
141                               (void *)0,                                 // pdata not used here
142                               &TaskStartStk[TASK_STK_SIZE-1],            // p_tos(top of stk)
143                               TASK_3_PRIO,                               // Task priority
144                               TASK_3_PRIO,                               // Task ID
145                               &TaskStartStk[0],                          // p_bos
146                               sizeof(AppTask1Stk)/sizeof(OS_STK),        // n of stack words
147                               (void *)p_tsk_ext,                         // p_Context info
148                         0);                                              // Task options
149
150        OSprintf ("\n\r Starting AppStartTask infinite loop" );
```

```
       Jan 24 20:27 2000  uCOS2test.c  Page 4


151       while (TRUE) {                 /* Task body, always written as an infinite loop. */
152           if (first_time)
153           {
154               first_time=FALSE;
155           }
156           /*---- Task code goes HERE! ------------------------------------------------------*/
157           OSTimeDly(1);                /* Delay task execution for one clock tick   */
158       }
159   }
160
161 /*$PAGE*/
162 /*
163 *********************************************************************************************
164 *                                          TASK #1
165 *
166 * Description : This is an example of a task.
167 * Arguments   : pdata   is the argument passed to 'AppTask1()' by 'OSTaskCreate()'.
168 * Notes: 1) The first line of code is used to prevent a compiler warning because 'pdata' is no
169 *     not used.  The compiler should not generate any code for this statement.
170 *     2) Interrupts are enabled once the task start because the I-bit of the CCR register was
171 *                  set to 0 by 'OSTaskCreate()'.
172 *********************************************************************************************
173 */
174
175 static void  AppTask1 (void *pdata)
176 {
177     INT32U Task1Ctr=0;
178     static char    q[5];
179                                          // Task initialization code goes HERE!
180     q[0]='-'; q[1]='\\'; q[2]='|'; q[3]='/';
181     OSprintf ("\n\r Task 1, TaskCtr = %d ", Task1Ctr);
182     while (TRUE)                          // Task body, always written as an infinite loop.
183     {                                     // Task code goes HERE!
184         Task1Ctr++;
185         OSprintf ("\r Task1 pass %d    %c", Task1Ctr, q[Task1Ctr&3]);
186         OSTimeDlyHMSM(0, 0, 2,0);        // Delay task execution for 1 second.
187         P0_OUT ^= 0x80;
188     }
189 }
190
191 static void  AppTask2 (void *pdata)
192 {
193     INT32U Task2Ctr=0;
194     INT8U  error;
195                                          // Task initialization code goes HERE!
196     OSprintf
197    ("\n\r Task 2 now waiting for an ASC0 interrupt to release the semaphore. \
198 Pass # %d\n\r",Task2Ctr);
199    while (TRUE)                          // Task body, always written as an infinite loop.
200    {                                     // Task code goes HERE!
```

```
       Jan 24 20:27 2000  uCOS2test.c  Page 5


201
202           Task2Ctr++;
203
204           OSSemPend(RxSem, 10000, &error);   // Do a "P" primitive on the semaphore.
205
206           OSprintf ("\n\r Task 2 waken (pass # %d) up by a receive interrupt or semaphore timed
out \n\r",Task2Ctr);
207
208           OSTimeDly(1);                      //  Delay task execution for one clock tick
209       }
210   }
```

7.0 BIBLIOGRAPHY

MicroC/OS-II, The Real-Time Kernel
Jean J. Labrosse
R&D Books
ISBN 0-87930-543-6

The C PROGRAMMING LANGUAGE (ANSI)
Brian W.Kernigan, Dennis M.Ritchie second edition.
Prentice Hall
ISBN 0-13-110362-8

Communication of the ACM, Vol. 11, No. 5
May 1968, pp. 345-346 Copyright © 1968
Related to Edsger W. Dijkstra who described the use of semaphores in multitasking systems.

Multitasking-PASCAL extensions solve concurrency problems
EDN (magazine)
September 29, 1982, p145
This article introduces the use of semaphores for mutual exclusion.

TriCore Architecture Manual
Infineon Technologies
Contains information related to the TC10GP chip used on the "TriBoard", like memory map, peripheral register addresses, GPTU (timer), ASC (UART) programming information, and more.

TC10GP User's manual
Infineon Technologies

Triboard Hardware Manual
Infineon Technology (Munich)

Documentation related with Green Hills Software compiler, released as a set of `.PDF` files or documentation related to the assembler/compiler/linker that you are using.

8.0 Web Sites

The Infineon Technology TriCore web site:

http://www.infineon.com/tricore

You will find information on the latest TriCore architecture, user's manual and more on this web site.

The µC/OS-II web site:

http://www.uCOS-II.com

9.0 ACKNOWLEDGEMENTS

A special thanks to (in alphabetic order):

Robert Chyla at Signum Systems for the help and enhancements.

Jean J. Labrosse (Mister µC/OS) for promptly and patiently replying to my emails and clarifying the details of the kernel port. Jean, your enthusiasm is contagious.

The TriCore architecture team. In San Jose, to each and all of you for your continuous support, and patience: *Daniel*, *Jithendra*, *Karl Heinz*, *Ravi*, *Roger*, *Steve*. For your competent and friendly support, clarifying the details of the architecture. And for being there every time I needed your help to understand this new "beast" called **TriCore**.

The TriBoard design team in Munich. *Ewald* and *Holger* for helping with the TriBoard. Also thanks for some of the "seed code" which made the task easier. I hope you will use µC/OS-II and have fun with it.

Sudhir (Infineon, San Jose) for the opportunity. This was challenging and fun.

Steven Ginzburg at Green Hills Software for all the support and patience. You were very nice.

To all the people I must have forgotten in this, forgive me for the omission: just take this as an unintentional "oops".

Andre Gompel